Election Day

Welcome to Election Day on Exercism's Go Track. If you need help running the tests or submitting your code, check out `HELP.md`. If you get stuck on the exercise, check out `HINTS.md`, but try and solve it without using those first :)

Introduction

Like many other languages, Go has pointers. If you're new to pointers, they can feel a little mysterious but once you get used to them, they're quite straight-forward. They're a crucial part of Go, so take some time to really understand them.

Before digging into the details, it's worth understanding the use of pointers. Pointers are a way to share memory with other parts of our program, which is useful for two major reasons:

1. When we have large amounts of data, making copies to pass between functions is very inefficient. By passing the memory location of where the data is stored instead, we can dramatically reduce the resource-footprint of our programs.
2. By passing pointers between functions, we can access and modify the single copy of the data directly, meaning that any changes made by one function are immediately visible to other parts of the program when the function ends.

Variables and Memory

Let's say we have a regular integer variable `a`:

```go
var a int
```

When we declare a variable, Go has to find a place in memory to store its value. This is largely abstracted from us — when we need to fetch the value stored in that piece of memory, we can just refer to it by the variable name.

For instance, when we write `a + 2`, we are effectively fetching the value stored in the memory associated with the variable `a` and adding 2 to it.

Similarly, when we need to change the value in the piece of memory of `a`, we can use the variable name to do an assignment:

```
a = 3
```

The piece of memory that is associated with `a` will now be storing the value `3`.

Pointers

While variables allow us to refer to values in memory, sometimes it's useful to know the **memory address** to which the variable is pointing. **Pointers** hold the memory addresses of those values. You declare a variable with a pointer type by prefixing the underlying type with an asterisk:

```
var p *int // 'p' contains the memory address of an integer
```

Here we declare a variable `p` of type "pointer to int" (`*int`). This means that `p` will hold the memory address of an integer. The zero value of pointers is `nil` because a `nil` pointer holds no memory address.

Getting a pointer to a variable

To find the memory address of the value of a variable, we can use the `&` operator. For example, if we want to find and store the memory address of variable `a` in the pointer `p`, we can do the following:

```
var a int
a = 2

var p *int
p = &a // the variable 'p' contains the memory address of 'a'
```

Accessing the value via a pointer (dereferencing)

When we have a pointer, we might want to know the value stored in the memory address the pointer represents. We can do this using the `*` operator:

```go
var a int
a = 2

var p *int
p = &a // the variable 'p' contains the memory address of 'a'

var b int
b = *p // b == 2
```

The operation `*p` fetches the value stored at the memory address stored in `p`. This operation is often called "dereferencing".

We can also use the dereference operator to assign a new value to the memory address referenced by the pointer:

```go
var a int        // declare int variable 'a'
a = 2            // assign 'a' the value of 2

var pa *int
pa = &a          // 'pa' now contains to the memory address of 'a'
*pa = *pa + 2    // increment by 2 the value at memory address 'pa'

fmt.Println(a)   // Output: 4
                 // 'a' will have the new value that was changed via the pointer!
```

Assigning to `*pa` will change the value stored at the memory address `pa` holds. Since `pa` holds the memory address of `a`, by assigning to `*pa` we are effectively changing the value of `a`!

A note of caution however: always check if a pointer is not `nil` before dereferencing. Dereferencing a `nil` pointer will make the program crash at runtime!

```go
var p *int // p is nil initially
fmt.Println(*p)
// panic: runtime error: invalid memory address or nil pointer dereference
```

Pointers to structs

So far we've only seen pointers to primitive values. We can also create pointers for structs:

```
type Person struct {
    Name string
    Age  int
}

var peter Person
peter = Person{Name: "Peter", Age: 22}

var p *Person
p = &peter
```

We could have also created a new `Person` and immediately stored a pointer to it:

```
var p *Person
p = &Person{Name: "Peter", Age: 22}
```

When we have a pointer to a struct, we don't need to dereference the pointer before accessing one of the fields:

```
var p *Person
p = &Person{Name: "Peter", Age: 22}

fmt.Println(p.Name) // Output: "Peter"
                    // Go automatically dereferences 'p' to allow
                    // access to the 'Name' field
```

Slices and maps are already pointers

Slices and maps are special types because they already have pointers in their implementation. This means that more often than not, we don't need to create pointers for these types to share the memory address for their values. Imagine we have a function that increments the value of a key in a map:

```
func incrementPeterAge(m map[string]int) {
        m["Peter"] += 1
}
```

If we create a map and call this function, the changes the function made to the map persist after the function ended. This is a similar behavior we get if we were using a pointer, but note how on this example we are not using any referencing/dereferencing or any of the pointer syntax:

```
ages := map[string]int{
  "Peter": 21
}
incrementPeterAge(ages)
fmt.Println(ages)
// Output: map[Peter:22]
// The changes the function 'incrementPeterAge' made to the map are visible after the fur
```

The same applies when changing an existing item in a slice.

However, actions that return a new slice like `append` are a special case and **might not** modify the slice outside of the function. This is due to the way slices work internally, but we won't cover this in detail in this exercise, as this is a more advanced topic. If you are really curious you can read more about this in Go Blog: Mechanics of 'append'

Instructions

A local school near you has a very active students' association. The students' association is managed by a president and once every 2 years, elections are run to elect a new president.

In this year's election, it was decided that a new digital system to count the votes was needed. The school needs your help building this new system.

1. Create a vote counter

One of the first things that the new voting system needs is a vote counter. This counter is a way to keep track of the votes a particular candidate has.

Create a function `NewVoteCounter` that accepts the number of initial votes for a candidate and returns a pointer referring to an `int` , initialized with the given number of initial votes.

```
var initialVotes int
initialVotes = 2

var counter *int
counter = NewVoteCounter(initialVotes)
*counter == initialVotes // true
```

2. Get number of votes from a counter

You now have a way to create new counters! But now you realize the new system will also need a way to get the number of votes from a counter.

Create a function `VoteCount` that will take a counter (`*int`) as an argument and will return the number of votes in the counter. If the counter is `nil` you should assume the counter has no votes:

```
var votes int
votes = 3

var voteCounter *int
voteCounter = &votes

VoteCount(voteCounter)
// => 3

var nilVoteCounter *int
VoteCount(nilVoteCounter)
// => 0
```

3. Increment the votes of a counter

It's finally the time to count the votes! Now you need a way to increment the votes in a counter.

Create a function `IncrementVoteCount` that will take a counter (`*int`) as an argument and a number of votes, and will increment the counter by that number of votes. You can assume the pointer passed will never be `nil` .

```
var votes int
votes = 3

var voteCounter *int
voteCounter = &votes

IncrementVoteCount(voteCounter, 2)

votes == 5          // true
*voteCounter == 5   // true
```

4. Create the election results

With all the votes now counted, it's time to prepare the result announcement to the whole school. For this, you notice that having only counters for the votes is insufficient. There needs to be a way to associate the number of votes with a particular candidate.

Create a function `NewElectionResult` that receives the name of a candidate and their number of votes and returns a new election result.

```go
var result *ElectionResult
result = NewElectionResult("Peter", 3)

result.Name == "Peter"  // true
result.Votes == 3       // true
```

The election result struct is already created for you and it's defined as:

```go
type ElectionResult struct {
    // Name of the candidate
    Name    string
    // Votes of votes the candidate had
    Votes   int
}
```

5. Announce the results

It's time to announce the new president to the school! The president will be announced in the little digital message boards that the school has. The message should show the name of the new president and the votes they had, in the following format: `<candidate_name> (<votes>)`. This is an example of such message: `"Peter (51)"`.

Create a function `DisplayResult` that will receive an `*ElectionResult` as an argument and will return a string with the message to display.

```go
var result *ElectionResult
result = &ElectionResult{
    Name: "John",
    Votes: 32,
}
```

```go
DisplayResult(result)
// => John (32)
```

6. Vote recounting

To make sure the final results were accurate, the votes were recounted. In the recount, it was found that the number votes for some of the candidates was off by one.

Create a function `DecrementVotesOfCandidate` that receives the final results and the name of a candidate for which you should decrement its vote count. The final results are given in the form of a `map[string]int` , where the keys are the names of the candidates and the values are its total votes.

```go
var finalResults = map[string]int{
    "Mary":  10,
    "John":  51,
}

DecrementVotesOfCandidate(finalResults, "Mary")

finalResults["Mary"]
// => 9
```

Source

Created by

- @andrerfcsantos